Mathematical Modeling of a Parallel Global Optimization Algorithm

Elizabeth Eskow *
Robert B. Schnabel *

CU-CS-395-88 April 1988

Department of Computer Science
Campus Box 430
University of Colorado,
Boulder, Colorado, 80309
U.S.A.

Research supported by AFOSR grant AFOSR-85-0251, and NSF cooperative agreement DCR-8420944.

Abstract

We describe the formation of a mathematical model of a reasonably complex parallel global optimization program, and the use of this model to assist in the development and understanding of the underlying parallel algorithm. First we discuss the formation of a model that accurately matched execution times of the parallel program on an Intel hypercube. Then we discuss the use of this model to simulate the behavior of our parallel algorithm in a variety of new situations, in order to detect weaknesses in the parallel algorithm and analyze possible improvements to it. We believe that this combination of parallel computer implementation and mathematical modeling is a useful approach in parallel algorithm development.

1. Introduction

We discuss the development and use of a mathematical model of a parallel global optimization algorithm. This model was formulated using actual timings of the algorithm on an Intel hypercube. It was then used to predict the behavior of the algorithm in various situations in order to understand the limitations of the algorithm, to suggest improvements to it, and evaluate some possible improvements, without implementing and running all these possibilities on the computer. We found that a fairly simple model was sufficient to provide a reasonable fit to a fairly complex parallel algorithm, and that using this model not only saved us considerable computer time, but also provided useful new insights about the parallel optimization algorithm. We believe that this approach is a useful one for parallel algorithm development, and that our experiences may be of interest to other parallel algorithm developers.

The basic problem addressed by this research is that for any given application, parallel algorithm developers are faced with a vast number of combinations of parallel algorithms, parallel architectures, and choices of number of processors that they would like to evaluate. The number of combinations is far too large to evaluate all of them through exhaustive computer testing. On the other hand, mathematical analysis of the algorithms, using counts of operations including arithmetic, communication, and others, may be too detailed a tool for sizeable problems. The approach we describe here is a combination of these two methods. First we use the results of a relatively small number of parallel computer experiments to create a rather high level model of our parallel algorithm. Then we use this model to simulate the behavior of our parallel algorithm in a variety of new situations, and use these simulations to better understand the method, suggest improvements to it, and evaluate some of these possible improvements. This

may, at some point, lead to the need for additional computer tests. We feel that this process leads to more efficient development of parallel algorithms, and to better understanding of the methods, than is obtained through the exclusive use of computer testing.

We have chosen to study the global optimization problem in this paper for two reasons. First, we have considerable experience and interest in developing parallel methods for this problem (see e.g. Byrd, Dert, Rinnooy Kan, and Schnabel [1]). Second, the level of complexity of this problem seems well suited to serve as a test case for our modeling approach. The parallel algorithm is non-trivial, having several distinct sections that must be modeled separately, but it is not so complex that its description and model will overwhelm the discussion. Another interesting aspect of the algorithm is that it has both deterministic and non-deterministic sections; it will be seen that the latter present interesting challenges for our approach.

The type of approach we describe has recently been advocated by others, for example Reed [5]. To our knowledge, however, few case studies of this type of approach have been described in the literature. We were especially influenced by the use of related modeling techniques by Hachtel and Moceyunas [3] applied to parallel algorithms for problems arising in VLSI design.

In Section 2, we briefly describe the parallel global optimization problem and the parallel algorithm for it that we will model. Section 3 discusses the formulation of the mathematical model of this algorithm, and points out the interesting issues we faced in constructing this model. Section 4 describes several applications of the model as a tool for understanding the algorithm's performance better and for predicting the effects of algorithmic improvements. In Section 5 we briefly draw several conclusions.

2. The Parallel Global Optimization Method

The global optimization problem is to find the lowest minimizer in a region D of a function $f(x)$ of n variables that has multiple local minimizers, lowest points of the function in some open subregion of D. This problem occurs in many practical applications, and may be very expensive to solve, requiring many evaluations of $f(x)$ and many local minimizations to be performed. Thus is it desirable to construct parallel methods that significantly speed up the solution of these expensive problems. The work described in this paper was initiated as a continuation of the research of Byrd, Dert, Rinnooy Kan, and Schnabel [1], to improve the parallel global optimization method described there.

The parallel global optimization method described in Byrd et al is related to the iterative, stochastic sequential methods developed by Rinnooy Kan and Timmer [6]. In the sequential method, at each iteration sample points are randomly generated over the problem domain, a bounded region D. Then a subset of those sample points are selected as start points for local minimizations, by the following procedure. A sample point is selected as a start point if and only if it has the lowest function value of all sample points within a certain "critical distance" of it. That is, the start points are, in some heuristic sense, the local minimizers of $f(x)$ over the sample. (The critical distance is computed by the algorithm as a function of sample size, iteration, number of variables, and other factors, as suggested by the analysis of the method.) Then a standard local minimization algorithm is applied from each start point, terminating each time at some local minimizer. The goal is that each of these local minimizers is distinct, and that each local minimizer is found exactly once. A probabilistic stopping rule is then applied to determine whether or not to perform another iteration. If so, the process is repeated with the new sample points being

added to the existing sample. If not, the lowest local minimizer found is reported as the global minimizer.

The parallel global optimization method of Byrd et al retains the basic structure of this sequential algorithm, and parallelizes each of its main steps. First it divides the domain D equally among the processors. Each processor generates sample points in its own subregion of D, and then tentatively selects some of these sample points to be candidate start points for local minimizations, by the same process as in the sequential method. Next, any of these candidate start points that are within the critical distance of a subregion border are communicated to neighboring subregions, and are removed as start points if a neighboring subregion contains a lower start point within the critical distance of the candidate. All remaining start points are then collected centrally, and distributed one to each processor to perform the local minimizations. If there are more start points than processors, each processor is assigned another start point as soon as it completes its current minimization, until all start points have been processed. This method is outlined in Algorithm 2.1 below.

As we will see more clearly in Section 3, the method just described requires fairly little communication or synchronization. The main needs for communication are to distribute candidate start points that are near borders in Step 2, and to collect and distribute the start points for Step 3. In each case, fairly little information is involved. Thus the algorithm appears well suited to shared or distributed memory multiprocessors.

Test results for an implementation of this algorithm on a network of 4 and 8 Sun workstations are presented in Byrd et al. We implemented a very similar algorithm on a 32 node Intel hypercube in order to compare the results in these two distributed memory multiprocessor

---

Algorithm 2.1 -- A Parallel Global Optimization Algorithm

Given $f : R^n \rightarrow R$, hyper-rectangle D, p processors

At each iteration :

1. **Generate sample points and function values**
   Partition D into p subregions, one per processor
   Each processor randomly chooses $\frac{s}{p}$ additional sample points in its subregion and evaluates $f(x)$ at each new sample point.

2. **Select start points for local minimizations**
   Each processor selects start points in its subregion (any sample point for which there is no lower sample point in its subregion within the critical distance).
   Resolve start points near borders between subregions (start point is removed if there is a lower sample point within critical distance in neighboring subregion).

3. **Perform local minimizations from all start points**
   Collect all start points and distribute one to each processor, which performs local minimization from that point.
   If there are more than p start points, distribute a remaining start point to each processor as soon as it completes its current local search. (Stop when local searches have been completed from all start points.)

4. **Decide whether to stop**
   Decide whether to stop using stochastic stopping rules.
   If stopping rules are satisfied, report lowest minimizer found as global minimizer.
   If stopping rules not satisfied, begin next iteration.

---

environments, and to obtain timings of the algorithm for a larger number of processors. The hypercube test results that we used in the creation of the model are for 3 standard test functions. Each problem was run with all combinations of sample sizes 200 or 1000, on 4, 8, 16 or 32 nodes of the hypercube, and with inexpensive or expensive function evaluation. Thus there are 16 variants of each problem, 48 test cases overall. The raw data is contained in Table 3.1. The test problems all have $n = 2$ or 4; this reflects the small size of current global optimization test problems. All the problems required only one iteration of Algorithm 2.1.

The reason for using both inexpensive and expensive function versions of the test problems is that both types of problems are important in practice, and the behavior of the parallel algorithm can vary significantly depending whether the function is expensive or not. The standard test problems have inexpensive functions, typically requiring around 0.001 - 0.01 second to evaluate on a node of the Intel hypercube. To make them expensive, we defined a function evaluation as either 100 or 1000 evaluations of $f(x)$, with the factor chosen so that the function evaluation cost is about a second. An interesting question is "what is an expensive function evaluation in the context of this algorithm?"; in Section 4 we will use our model to answer this question, and will see that the choice of 1 second is appropriate.

3. Formation of the Model

We formulated the mathematical model of our parallel global optimization algorithm in two stages. First, for each of the three main steps of the algorithm, we fit the 48 run times for this step in order to determine what terms to include, and omit, in the model for this step. Then we determined the unknown parameters in the entire resultant model by fitting the overall model to the 48 overall runs times. In this section we summarize this process and discuss its interesting aspects.

The independent variables that are required to model the performance of the algorithm clearly include the number of processors p, the number of variables n, the total sample size per iteration s, and the cost of an evaluation of $f(x)$, f. In addition, to model the local search phase of the algorithm, two more independent variables are required which cannot be predicted as functions of the other independent variables. These are the number of function evaluations, *fcns*, and local search iterations, *itns*, of the processor whose search phase was longest. These are usually

equal to the number of function evaluations and iterations, respectively, in the longest local search, because the number of searches is typically less than p. (Typically *fcns* $\in [n+1, n+2]$ *itns*.)

The costs of Step 1 of Algorithm 2.1 include the random generation of the sample points, the function evaluations of the sample points, and some overhead, which are readily seen to require $\frac{s}{p} n \cdot c_{11}$, $\frac{s}{p} f$, and c_{12} operations, respectively. (Unknown parameters for the individual phases are denoted by c_{ij}, where i is the step number in Algorithm 2.1.) By inspecting the runs times for Step 1 it was clear that the times for random number generation and the overhead cost were insignificant in comparison to the cost of evaluating the function at all sample points. Hence, the only term used from Step 1 is $\frac{s}{p} f$. Note that no unknown parameter is involved.

Step 2 is the most difficult portion of the algorithm to model because there is so much variation possible in the amount of work required to select start points for local minimizations from a particular size sample. This is due to the nondeterministic nature of the start point selection algorithm, as discussed below.

The costs for Step 2 include an initial sort by function value of the points within each subregion, the comparison of sample points within each subregion to find the candidate start points (points which have a lower function value than any other sample point in the subregion within the critical distance from them), and finally the comparison of candidate start points near subregions boundaries to sample points from neighboring subregions. To increase the efficiency of these steps, the domain is subdivided into $K \geq p$ subregions, where K is fixed, so that each processor handles K/p contiguous subregions. This reduces the worst case complexity of the candidate start point selection in Step 2 by a factor of K/p, and makes a substantial difference in

practice. As a result, the border resolution is performed in two stages, first resolving with subregions on the same processor and then resolving with subregions from different processors.

The cost of the initial sort is $\frac{s}{p} log\, s \cdot c_{21}$. This step was clearly insignificant in comparison to the remainder of Step 2 and so it was omitted from the model.

To select candidate start points, each sample point x_s in the subregion is compared to each other point x_c in the subregion that has a lower function value, to see if x_c is within the critical distance of x_s. As soon as such an x_c is found then x_s can not be a candidate start point and the algorithm goes on to the next sample point; if no such x_c is found then x_s becomes a candidate start point. Each comparison requires $O(n)$ operations. Thus the worst case complexity of this step is $\frac{s^2}{p} n \cdot c_{22}$ in the case that each x_s must be compared to all lower points in the subregion. On the other hand, the best case would be $\frac{s}{p} n \cdot \hat{c}_{22}$ in the case that each x_s must be compared to only one point. We chose the worst case term in generating the model because it gave the better fit among the two possibilities, but as should be expected, the fit for this phase is not as good as the fits for other steps of the algorithm.

The remaining portion of Step 2 is the comparison of candidate start points near subregions boundaries to sample points in neighboring subregions. As mentioned above, this proceeds in two stages, with subregions on the same processor and then with subregions on other processors. Of these two phases, the cost for resolving points with subregions on different processors was clearly the dominant term and was used in the model. This phase also is nondeterministic; each processor might have to compare between p and s candidate start points, to between 1 and $\frac{s}{p}$ of its own points. The most representative case in practice seemed to be the comparison of $O(p)$

border points to $O(\frac{s}{p})$ points each, which simplifies to $s\ n \cdot c_{23}$. Again this analysis is not always accurate in modeling this phase of the algorithm. This phase also includes some interprocessor communication which we discuss below.

Step 3 of Algorithm 2.1 is easier to model. The costs of this step include the function evaluations, the linear algebra calculations associated with each iteration, and an overhead cost. The term for the function evaluations of the search phase is $fcns \cdot f$. The cost per iteraton includes a constant term and terms that are linear and quadratic in n. We found it necessary to use only the quadratic term, i.e. $itns \cdot n^2 \cdot c_{32}$. The overall overhead term and the term which was linear in n for each iteration were found to be insignficant, and were not included in the model.

The only cost of Algorithm 2.1 that remains to be discussed is the interprocessor communication. The amount of communication required by the algorithm is minimal. In each iteration there are 6 messages between the master process (located on the Intel Hypercube cube manager) and the node processes, and 5 $lg\,p$ internode messages. On the Intel Hypercube, messages from the cube manager to nodes take approximately 6 times longer than node to node messages. An upper bound on the average message length for this algorithm is approximately 1500 bytes. Using these figures and the known message speeds for the Intel hypercube, an upper bound for the total time for interprocessor communication is approximately 0.23 seconds per iteration. By inspection of the times in Table 3.1, it can be seen that this term is small in comparison to the overall running times, and can be regarded as a constant. Thus it was combined with the other overhead costs of the algorithm to obtain the final term in the model, a single constant for the overhead.

Using the dominant terms from the individual phases as discussed above, we have modeled an iteration of Algorithm 2.1 by

$$\frac{s}{p} f + \frac{s^2}{p} n \cdot c_1 + s \cdot n \cdot c_2 + f \cdot fcns + itns \cdot n^2 \cdot c_3 + c_4 \tag{3.1}$$

where $c_1, \cdots, c_4$ are renumbered values of the unknown parameters. The first term models the leading cost of Step 1. The second term models the leading cost of the candidate start point selection phase of Step 2. The third term is a rough approximation to the border resolution phase of Step 2. The fourth and fifth terms model the leading costs of Step 3. The final term is an aggregate term for overhead costs in each step, and for interprocessor communication. As indicated above, there was no significant advantage was obtained by trying to model communication costs more carefully.

We fit this model of the entire algorithm to the 48 measurements of the test problems discussed previously. The function evaluation cost ranged from about 0.001 to 5 seconds for the 6 test functions used. Since this causes the execution times to vary widely, we fit the *relative* difference between the measured computer times and the model prediction, using linear least squares. The parameter values obtained were

$$c_1 = 0.000016\,,\ c_2 = 0.0035\,,\ c_3 = 0.010\,,\ c_4 = 0.49 \tag{3.2}$$

and resulted in a median relative error of 0.05 with a range of 0.002 to 0.19 on the 48 problems. Table 3.1 shows the measurements of the test problems, as well as the relative errors of the model at each data point. We find this fit of the data to the model to be sufficient for the uses that we wish to make of the model, which are discussed in the next section.

Table 3.1 -- Measured Times and Model Times for Test Problems (in seconds)

Problem	n	f	s	p	Measured Time	Model Time	Relative Error
BR	2	.00165	200	4	2.4	2.5	.048
			200	8	2.0	2.3	.162
			200	16	2.0	2.2	.106
			200	32	2.6	2.2	.166
			1000	4	16.0	16.2	.013
			1000	8	10.8	12.0	.106
			1000	16	10.0	9.8	.018
			1000	32	8.6	8.8	.018
GP	2	.00115	200	4	2.6	2.6	.002
			200	8	2.4	2.4	.011
			200	16	2.4	2.3	.030
			200	32	2.8	2.3	.185
			1000	4	17.8	16.2	.092
			1000	8	13.2	12.0	.094
			1000	16	10.8	9.9	.086
			1000	32	9.4	8.8	.062
S5	4	.0486	200	4	18.8	17.9	.049
			200	8	16.4	16.3	.004
			200	16	15.2	15.6	.025
			200	32	14.8	15.2	.026
			1000	4	48.0	48.7	.014
			1000	8	32.0	33.4	.042
			1000	16	23.8	26.2	.100
			1000	32	21.6	22.7	.052
EXPBR	2	1.49	200	4	112.4	108.2	.038
			200	8	74.8	70.8	.054
			200	16	56.6	52.1	.080
			200	32	47.8	42.7	.106
			1000	4	434.2	427.0	.017
			1000	8	242.6	236.7	.024
			1000	16	148.0	141.5	.044
			1000	32	100.0	94.0	.060
EXPGP	2	1.53	200	4	136.2	131.0	.038
			200	8	99.4	94.2	.053
			200	16	79.2	73.4	.073
			200	32	71.6	65.3	.087
			1000	4	467.8	453.4	.030
			1000	8	263.6	258.1	.020
			1000	16	165.0	160.5	.028
			1000	32	116.2	111.6	.039
EXPS5	4	4.84	200	4	1027.2	971.4	.054
			200	8	901.0	850.0	.057
			200	16	840.4	789.4	.060
			200	32	810.0.0	759.1	.063
			1000	4	1795.0	1649.0	.081
			1000	8	1033.4	953.3	.078
			1000	16	695.6	637.1	.084
			1000	32	558.6	493.5	.117

4. Applications of the Model

We have used the model (3.1-2) to simulate the behavior of our parallel algorithm in a variety of situations. These simulations have allowed us to understand our algorithm better, to explore its limitations, and to assess posssible modifications to the algorithm. This section summarizes several such uses.

One major way we have used our model is to help understand the limitations of our parallel implementation. We did this by tabulating the contributions of each of the six terms of the model, for a representative sample of problems. For example, Table 4.1 shows the run times predicted by the model for the 8 problems that have all possible combinations of $f = 0.001$ or 1, $s = 200$ or 1000, $p = 8$ or 32, with $n = 3$, $itns = 20$, and $fcns = 100$.

Even though we had already worked with this parallel algorithm for a long time, we learned a considerable amount from tabulations like Table 4.1. When we began this work, the start point selection used p as opposed to $K > p$ subdivisions. Tabulations of the model at that point made it very clear that the $O(s/p)^2$ cost of this approach was unacceptable for large s/p, so we changed the method to use K subdivisions as explained in Section 3. Now that we have made that change (and revised the model), our biggest surprise with Table 4.1 concerns the border resolution phase. We recognized that this phase, although parallel, does not speed up with p, but the cost of this phase was thought to be insignificant. The model tabulation results shows that this phase is significant for large values of p and s and small values of f. Consequently we have begun investigating better approaches to this phase.

Other effects of varying the parameter values were more predictable. As the cost of function evaluations increase, the local search and sample generation parts of the algorithm dominate.

When f is inexpensive, the start point selection phase becomes significantly more important. Increasing the dimension of the problem, n, would not change the relative costs of the various steps much, since n is basicly linear throughout the model. Table 4.1 also confirms that the overhead of our parallel algorithm, including communication, is small even for the cheapest problems.

Tabulations like Table 4.1 can also be used to answer simple questions about the performance of the algorithm on different parallel computers. We could use the model to extrapolate to larger values of p and see when communication would become a bottleneck. We could also use the model to study the effect of implementing the algorithm on other distributed memory machines. The main change would be the cost of communication relative to floating point speed, and the effect of this change could be determined by modifying the part of the overhead term dealing with communication (roughly half the overhead term).

Another use we made of the model was to help determine how expensive function evaluations must be to dominate the other costs of the algorithm. Many parallel and sequential optimization studies consider only the cost of function evaluations, on the assumption that this is the dominant cost, but they almost never say what the threshhold for "expensive" is. We used our model to determine the value of f for which the cost of functions evaluations (terms 1 and 4) would be 10 times the sum of all the other terms, for various combinations of realistic values of the other parameters. Table 4.2 shows some representative tabulations. In all cases, the threshhold value of f was between 0.4 and 4 seconds, or roughly 1000 flops. This shows that for even moderately expensive functions, speedups estimated using only function evaluations are off by at most 10%.

The last use of the model that we will discuss is to predict the results of proposed improvements to the algorithm. We wanted to be able to determine the expected gains in efficiency of proposed improvements to the algorithm prior to implementing and running them on a parallel computer. The main improvement that we have considered is parallelizing each local search by performing a speculative evaluation of the finite difference gradient (Schnabel [7]) in conjunction with each function evaluation. This means that the n function evaluations for the finite difference gradient are performed in parallel with the standard function evaluation, before it is known whether this information will be required. This modification enables each local search to utilize up to $n+1$ processors as opposed to one processor in Algorithm 2.1. Most of the time the gradient is required and the speculative work is worthwhile, while roughly 20% of the time the gradient is not needed and the speculative work is wasted.

We have modified the model to account for this change to the algorithm. The sample generation, start point selection, border resolution, linear algebra cost per search iteration, and the original overhead costs are unaffected by this change, so these terms in the model remain the same. The alterations are in the costs of function evaluations and communication. Let w be a new independent variable representing the total number of local searches performed, and assume that the average number of function evaluations per iteration of a local search (excluding the gradient evaluation) is 1.25, which corresponds to 20% of the speculative gradients being unnecessary. Then the total number of function evaluations per processor in the new algorithm is readily seen to be $1.25 \cdot itns \cdot \left\lceil \frac{(n+1)w}{p} \right\rceil$. The modification also introduces the need for $\lceil lg\,(n+1) \rceil$ extra messages per parallel gradient evaluation, so that the additional communication cost is estimated to be $1.25 \cdot \text{itns} \cdot \lceil lg\,(n+1) \rceil \cdot .004$ seconds. Thus Algorithm 2.1 modified by the specu-

lative gradient evaluations can be modeled by

$$\frac{s}{p} f + \frac{s^2}{p} n \cdot c_1 + s \cdot n \cdot c_2 + f \cdot 1.25 \cdot itns \cdot \left\lceil \frac{(n+1)w}{p} \right\rceil$$

$$+ itns \cdot n^2 \cdot c_3 + 1.25 \cdot itns \cdot \lceil lg(n+1) \rceil \cdot .004 + c_4 \ . \qquad (4.1)$$

An important point is that $c_1, \cdots, c_4$ can reasonably be assumed to have the same values as previously discussed, so that (4.1) can be used to predict the performance of the modified algorithm without doing any computer experiments.

Table 4.3 shows the speedups predicted by equation (4.3) of the new speculative gradient algorithm in comparison to Algorithm 2.1, on the same 8 problems as were used in Table 4.1, assuming that $w = 4$. In general, the models show that the extra communications overhead involved in doing the speculative gradients would cause a slight degredation in performance for very inexpensive functions. Once function evaluations require even 0.01 - 0.1 second, however, the model shows that the additional communication would become insignificant, substantial speedups would result in the cases where p is greater than $n+1$ times the number of local minimizations, and some speedup would occur as long as p is greater than or equal to $n+1$.

Finally, we mention that we investigated the reliablility and extensibility of our model by choosing several new problems, and comparing their computer timings to the values predicted by our model. We chose several problems that have either more minimizers or a larger number of variables than the problems on which the model was based. The problems with many minimizers are from Levy and Gomez [4], while the problems with different values for n are the two Hartman functions from Dixon and Szego [2]. Table 4.4 shows the timings for these problems compared to the model predictions. The fits for the problems with many minimizers are roughly as good as for the original test prolems, demonstrating that our model has no special difficulty with

this extension. The results for the Hartman 6 inexpensive function show, however, that our model does not scale particularly well with increasing n; this is due to the inaccuracies inherent in using the worst case term to model the nondeterministic cost of start point selection. In general, the results show that modeling expensive functions is more accurate than modeling cheap functions. This is to be expected because in this case the deterministic parts of the algorithm, which are much easier to model, dominate.

Table 4.1 -- Model Performance Predictions
($n = 3, itns = 20, fcns = 100$)

f	s	p	Sample Generation	Start Pt Selection	Border Resolution	Local Searches	Overhead
.001	200	8	0.03	0.24	2.1	1.54	0.49
.001	200	32	0.01	0.06	2.1	1.54	0.49
.001	1000	8	0.13	6.08	10.5	1.54	0.49
.001	1000	32	0.03	1.52	10.5	1.54	0.49
1.0	200	8	25.0	0.24	2.1	101.44	0.49
1.0	200	32	7.0	0.06	2.1	101.44	0.49
1.0	1000	8	125.0	6.08	10.5	101.44	0.49
1.0	1000	32	32.0	1.52	10.5	101.44	0.49

Table 4.2 -- Value of f for Which Function Evaluation Costs Equal 10 Times All Other Costs
($p = 32, fcns = n + 1 \cdot itns$)

n	s	$itns$	Predicted f
3	200	10	0.77
10	200	10	0.42
3	1000	10	1.53
10	1000	10	1.21
3	200	30	1.89
10	200	30	1.00
3	1000	30	3.59
10	1000	30	1.95

Table 4.3 -- Predicted Time for Parallel Gradient Algorithm (Same problems as Table 4.1)

f	s	p	Algorithm 2.1 Predicted Time	Parallel Gradient Algorithm Predicted Time	Predicted Speedup
.001	200	8	4.8	5.0	0.97
.001	200	32	4.6	4.7	0.97
.001	1000	8	19.1	19.3	0.99
.001	1000	32	14.5	14.6	0.99
1.0	200	8	129.7	79.9	1.62
1.0	200	32	111.5	36.7	3.04
1.0	1000	8	243.9	194.1	1.27
1.0	1000	32	146.4	71.6	2.05

Table 4.4 -- Measured Times vs Model Predictions for Other Functions (s = 1000)

Problem	n	f	p	Measured Time	Model Time	Relative Error
LG#4	2	.016	8	18.6	15.2	.183
			32	8.8	9.8	.114
EXPLG#4	2	1.6	8	343.8	324.1	.057
			32	121.8	111.2	.087
LG#8	3	.016	8	39.6	31.3	.210
			32	17.8	17.0	.045
EXPLG#8	3	1.8	8	978.0	902.0	.078
			32	349.2	261.1	.252
H3	3	.031	8	23.0	23.0	.001
			32	15.4	15.6	.013
EXPH3	3	3.7	8	660.2	643.1	.026
			32	307.4	290.7	.054
H6	6	.064	8	192.2	107.8	.439
			32	82.6	51.3	.379
EXPH6	6	6.9	8	5517.4	5255.6	.047
			32	2628.8	2162.3	.177

5. Conclusions

Mathematical modeling of a parallel global optimization algorithm based on actual parallel computer measurements has proven to be a useful technique in the continued development of the parallel algorithm. We found that it was possible to construct a fairly simple mathematical model

that modeled a rather complex parallel algorithm fairly well. Using this model to simulate additional computer runs of the algorithm on different problems or numbers of processors gave us considerable new insight into the algorithm's performance, and showed specifically where improvements could be made. The model was also modified to reflect possible changes to the algorithm, allowing us to assess the performance of these changes before implementing and running them. We believe that this process saved us considerable time, and gave us considerably more insight, than simply implementing numerous versions of the algorithm and running computer experiments on each. Our biggest surprise, given that we had developed the parallel algorithm and were already very familiar with it, was how many new and unexpected things we learned about it through the modeling process.

This work has also shown that there are challenges remaining in the area of accurately modeling algorithms. The main challenge that we encountered was trying to accurately model nondeterministic portions of the method. (Note that parallelism is not the problem, the same difficulties would exist if the algorithm were sequential.) In cases like ours where the worst case complexity is a gross overestimate, modeling using expected time would probably be preferable. But this introduces other difficulties: expected time complexity analysis often is quite difficult, and the unknown parameters of the model are likely to enter linearly rather than nonlinearly, necessitating nonlinear rather than linear least squares data fitting.

REFERENCES

[1] R.H. Byrd, C. Dert, A.H.G. Rinnooy Kan, and R.B. Schnabel, Concurrent stochastic methods for global optimization, Tech. Rpt. CU-CS-338-86, Dept. Comp. Sci., Univ. Colorado, 1986.

[2] L.C.W. Dixon and G.P. Szego, eds., *Towards Global Optimization 2* (North-Holland, Amsterdam, 1978).

[3] G.D. Hachtel and P.H. Moceyunas, Parallel Algorithms for Boolean Tautology Checking, *IEEE International Conference on Computer-Aided Design 1987 Digest of Papers,* (Santa Clara, California, 1987) 422-425.

[4] A.V. Levy and A. Montalvo, The tunneling method applied to global optimization, in: P.T. Boggs, R.H. Byrd and R.B. Schnabel, eds., *Numerical Optimization 1984* (SIAM, Philadelphia, 1985) 213-244.

[5] D.A. Reed, Parallel systems : performance modeling and numerical algorithms, presented at Third SIAM Conf. on Parallel Processing for Scientific Computation, 1987.

[6] A.H.G. Rinnooy Kan and G.T. Timmer, Stochastic methods for global optimization, *Am. J. Math. Mgmt. Sci.* **4** (1984) 7 - 40.

[7] R.B. Schnabel, Concurrent function evaluations in local and global optimization, Tech. Rpt. CS-CU-345-86, Dept. Comp. Sci., Univ. Colorado, 1986.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CU-CS-395-88	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION University of Colorado 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Air Force Office of Scientific Research/NM
6c. ADDRESS (City, State and ZIP Code) Computer Science Department Campus Box 430 Boulder, CO 80309-0430	7b. ADDRESS (City, State and ZIP Code) Building 410 Bolling AFB, DC 20332
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT NO.
11. TITLE (Include Security Classification) Mathematical Modeling of a Parallel Global Optimization Algorithm	
12. PERSONAL AUTHOR(S) Elizabeth Eskow and Robert B. Schnabel	
13a. TYPE OF REPORT Technical 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) 88/04/01 15. PAGE COUNT 19
16. SUPPLEMENTARY NOTATION	
17. COSATI CODES FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) Parallel computation, mathematical modeling, global optimization

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

We describe the formation of a mathematical model of a reasonably complex parallel global optimization program, and the use of this model to assist in the development and understanding of the underlying parallel algorithm. First we discuss the formation of a model that accurately matched execution times of the parallel program on an Intel hypercube. Then we discuss the use of this model to simulate the behavior of our parallel algorithm in a variety of new situations, in order to detect weaknesses in the parallel algorithm and analyze possible improvements to it. We believe that this combination of parallel computer implementation and mathematical modeling is a useful approach in parallel algorithm development.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ OTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Brian W. Woodruff, Major USAF	22b. TELEPHONE NUMBER (Include Area Code) 202/767-5025 22c. OFFICE SYMBOL